Modelling and managing deployment costs of microservice-based cloud applications

Leitner, Philipp ; Cito, Jürgen ; Stöckli, Emanuel

DOI: https://doi.org/10.1145/2996890.2996901

Modelling and Managing Deployment Costs of Microservice-Based Cloud Applications

Philipp Leitner and Jürgen Cito
Department of Informatics
University of Zurich
Zurich, Switzerland
{leitner|cito}@ifi.uzh.ch

Emanuel Stöckli
Institute of Information Management
University of St. Gallen (HSG)
St. Gallen, Switzerland
emanuel.stoeckli@unisg.ch

ABSTRACT

We present an approach to model the deployment costs, including compute and IO costs, of Microservice-based applications deployed to a public cloud. Our model, which we dubbed *CostHat*, supports both, Microservices deployed on traditional IaaS or PaaS clouds, and services that make use of novel cloud programming paradigms, such as AWS Lambda. *CostHat* is based on a network model, and allows for what-if and cost sensitivity analysis. Further, we have used this model to implement tooling that warns cloud developers directly in the Integrated Development Environment (IDE) about certain classes of potentially costly code changes. We illustrate our work based on a case study, and evaluate the *CostHat* model using a standalone Python implementation. We show that, once instantiated, cost calculation in *CostHat* is computationally inexpensive on standard hardware (below 1 ms even for applications consisting of thousand services and endpoints). This enables its use in real-time for developer tooling which continually re-evaluates the costs of an application in the background, while the developer is working on the code.

1. INTRODUCTION

The cloud computing paradigm [5] is by now a well-established concept enabling the flexible provisioning of IT resources, including computing power, storage and networking capabilities. While academic research on cloud computing has in the past primarily focused on backend and server-side issues, we are currently experiencing a surge of interest in cloud-based software development [7]. A key question in this domain is how to enable developers to build applications that are "cloud-native" [1], as opposed to applications that are merely migrated to cloud infrastructure. The Microservice architectural style [25] is nowadays commonly used as a baseline for building applications that are resilient towards cloud outages, and which are able to scale on a fine-granular basis, due to changes in the application workload or performance fluctuations. Fundamentally, Microservice-based applications are an implementation of the basic idea of service-oriented computing [13], whereas applications are built as a network of largely independent services communicating over standardized interfaces.

An implication of the on-demand, pay-as-you-go pricing model as it is commonly used in cloud computing is that predicting the monetary costs necessary to host a given application is often difficult. We have previously argued that today, deployment costs are generally not a tangible factor for cloud developers [8]. In Microservice-based cloud-native applications, this problem is even more prevalent. While a team responsible for building and operating a specific (set of) services may be aware of the operational costs of its own services, it is exceedingly difficult to estimate the cost impact that a change will have on other services in the application. For instance, a change in one service may lead to additional load on services that it depends on, causing those services to scale out. These effects are difficult to estimate during software development.

In this paper, we propose CostHat, a graph-based cost model for Microservice-based applications. We consider two different types of services, instance-backed services deployed either using Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) clouds, and Lambda services. We discuss how this cost model can be made actionable for cloud developers. Specifically, we discuss how the model can be used to conduct what-if analyses (e.g., analysing the cost impact of changes in the workload, such as an increased number of users), and to estimate the cost implications of new feature implementations or refactorings. We present a numerical evaluation of CostHat, which shows that evaluating costs in our model is computationally cheap on standard hardware, with calculation times remaining at or below 1 ms even for applications consisting of thousand services and endpoints. Our results show that the CostHat model can be implemented efficiently enough to foster real-time re-evaluation, which is necessary for the developer support tooling we envision.

The rest of this paper is structured as follows. In Section 2, we introduce the basic idea behind Microservices as well as CostHat. In Section 3, we discuss implementation issues of CostHat, and explain how developers can measure or predict the various parameters that go into CostHat. In Section 4, we explain different use cases for CostHat. In Section 5.1, we discuss a Python-based implementation of the model, an instantiation of the model for a case study application, as well as numerical experiments that show that the model can be re-evaluated quickly enough to support real-time analysis. Section 6 discusses related research, and finally Section 7 concludes the paper with lessons learned and an outlook on future work.

UCC'16, December 06-09, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/2996890.2996901

2. A GRAPH-BASED COST MODEL

We now introduce the fundamental CostHat model. We start by defining lambda-backed and instance-backed Microservices as the two fundamental service models supported by CostHat, followed by a deep-dive into the underlying formal model. We conclude the section with a recursive algorithm for workload and cost prediction.

2.1 Application Model

CostHat assumes applications to be based on a *Microservices* architecture [1, 25]. Such applications can be represented as a network of loosely-coupled services, which interact over well-defined remote interfaces. Each Microservice has limited scope and size, and can be implemented using whatever technology stack is suitable for its domain [26]. Each Microservice provides a specific, narrow business capability in the application. In this way, Microservices can be seen as a pragmatic implementation of the original idea of service-oriented computing [13]. An example of the high-level architecture of a simple Microservice-based application is given in Figure 1.

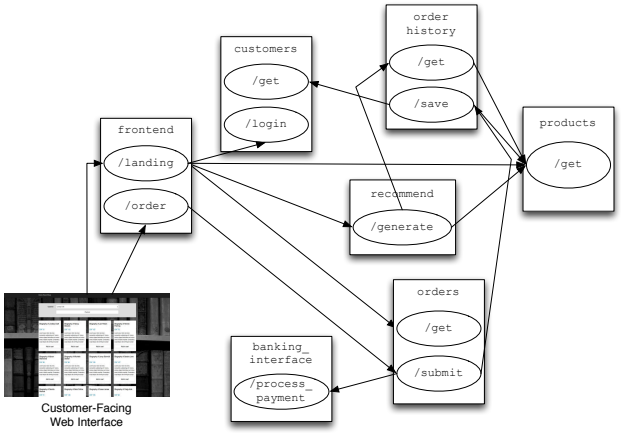

Figure 1: Snippet of the service architecture of a Microservice-based Web store. Services are defined along domain functionality, and provide one or more endpoints. Each service and endpoint interacts with multiple other services and endpoints to implement its functionality.

The Microservices paradigm is strongly tied to cloud computing as a deployment platform, as one of the fundamental motivations behind adopting Microservices is to enable fine-grained elasticity. That is, every Microservice forms its own deployment unit, which can be scaled independently of the rest of the application. A simplified "canonical" schematic view of a Microservice is given in Figure 2 (a). Each service has a fixed API consisting of one or more concrete endpoints (that is, functionality that the service is providing) and a data backend. To carry out the actual computation, and to implement autoscaling and elasticity, the service uses a pool of backing instances of varying size. This model can be implemented on top of IaaS or many PaaS clouds [22], with or without container technology, and using various data stores. Note that the model makes no assumption on the data store being one or more self-hosted databases, or a cloud data storage service, e.g., Amazon RDS. Additional value-added services (e.g., monitoring) are either implemented on a per-service level, or for the application as a whole, e.g., through common libraries. In the following, we refer to this canonical model as *instance-backed Microservices*.

In addition to the canonical model depicted in Figure 2 (a), we

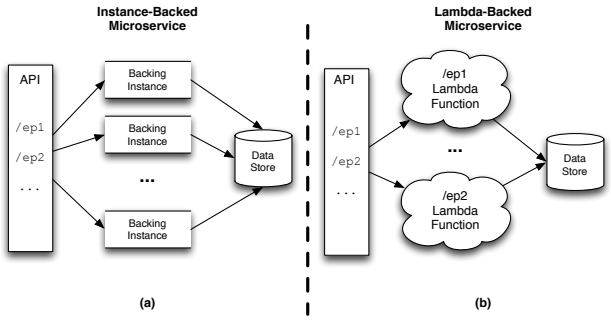

Figure 2: Two simple architectural views of a Microservice. (a) depicts a "canonical" service built on top of, for instance, an IaaS cloud, (b) shows a Microservice built on top of AWS Lambda or a comparable cloud service.

can also consider a second type of service, following the form depicted in Figure 2 (b). This model, which we dub *lambda-backed Microservice*, differs from instance-backed Microservices in that the backing instances are entirely invisible to the service developers (they are managed by the cloud provider completely transparently). This model represents a Microservice built on top of so-called "serverless" cloud services, for instance AWS Lambda[1]. However, this model can also be used to represent externally-hosted pay-per-use services, which can be integrated into a Microservice-based application.

2.2 Model Overview

CostHat models the costs of deploying and operating Microservice-based applications in a cloud. We define the total deployment costs of the application as the sum of operating each individual service in the application. The three primary drivers of service costs are the processing of domain functionality that the service is implementing, how many requests per time period it needs to accommodate, and in which quality (e.g., response time). Typically, the service quality is seen as a approximately constant, with the service scaling out to accommodate higher load at the expense of higher deployment costs, and vice versa for lower load.

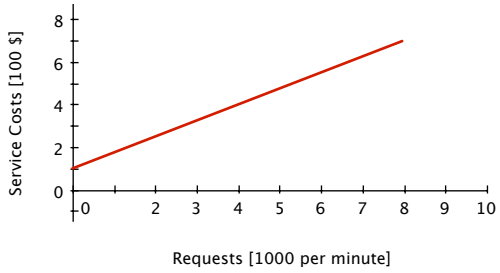

Figure 3: Schematic visualization of cloud cost for a lambda-backed Microservice depending on number of requests per time period it needs to handle.

Fundamentally, CostHat takes into account four cost factors for each service, (1) *compute costs* (payment for the CPU time necessary to process service requests), (2) *costs of API calls* (per-request payment for each API call), (3) *costs of IO operations* (payment for, for instance, database or file system writes), and (4) *costs of*

[1]http://docs.aws.amazon.com/lambda/latest/dg/welcome.html

additional options (e.g., payment for elastic IP addresses). The costs of additional options are generally constant, while the costs of API calls and IO operations are approximately linearly dependent on the number of requests that need to be handled. Finally, the compute costs for operating a service also depend on the number of requests. For lambda-backed Microservices, this dependency is linear as well. This leads to a cost function for lambda-backed Microservices similar to the one sketched in Figure 3, where the costs increase linearly with an increasing load on the service.

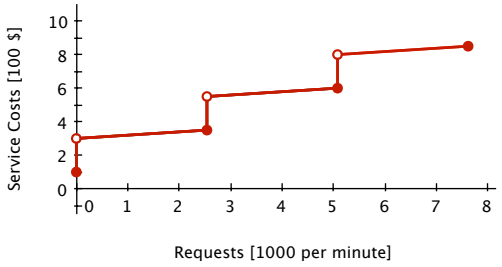

Figure 4: Schematic visualization of cloud cost for an instance-backed Microservice depending on number of requests per time period it needs to handle.

For instance-backed Microservices, costs for API calls, IO, and additional options remain as for lambda-based services. However, for compute costs, which are often the most important cost factor for operating cloud services, the connection between load and costs is non-linear (see Figure 4). The "jumps" in this cost function represent load thresholds where the service is required to scale out to an additional backing instance to keep the response time constant under increasing load.

2.3 Cost Impact of Changes

Given that services in a Microservice-based application are highly interconnected, changes in one service often impact the deployment of other services. Consider the example in Figure 5. An initial version of the `recommend` service delivers static recommendations only (e.g., a fixed "top-10 products of the day" list). Imagine now a change that lets this service deliver personalized recommendations based on previous orders instead. Hence, a new dependency to the `orderhistory` service is added, and for each request to the `recommend/generate` service endpoint, one or more additional requests to the `orderhistory/get` endpoint become necessary. Assuming that the `recommend` service will be used to render the landing page of the Web shop, this will lead to substantially increased load on the `orderhistory` service, likely requiring it to scale out to maintain its service level, hence increasing the operation costs of this service.

Even worse, as the `orderhistory/get` endpoint itself uses the `products/get` endpoint, the additional load may also require the `products` service to scale out as well. Given that individual Microservices are often built and operated by different teams [25], these cost effects of local development decisions are difficult to predict and manage. CostHat fundamentally aims to make these network effects explicit, and support what-if and change impact analysis for service developers, ultimately allowing them to reason about the impact that their changes have on other parts of the application at large (see Section 4).

2.4 CostHat

Based on these initial observations, we now define the CostHat cost model for Microservice-based applications. The two major in-

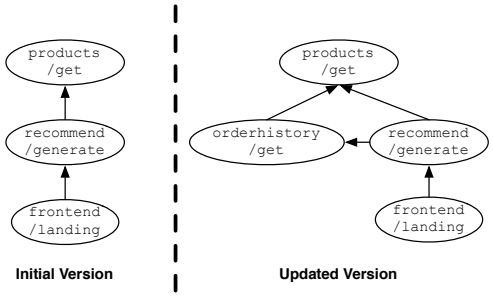

Figure 5: An update of the `recommend` service impacts the `orderhistory`, and transitively, the `products`, services.

gredients of CostHat are service call graph and service cost models, which we discuss and formalize in the following.

2.4.1 Service Call Graph Model

A Microservice-based application is fundamentally a collection of services $s \in S$. Each service in turn consists of concrete endpoints $\{s_{e1}, s_{e2}, \ldots s_{ei}\}, s_{ei} \in E$. Each service endpoint s_{ei} interacts with its consumers (either the end user, in the case of user-facing services, or other services) via requests $r \in R$. Each request has an originating service endpoint $s_{e,o} \in E$ as well as a target service endpoint $s_{e,t} \in E$.

Given these preliminaries, the workload w_s^ζ that a service needs to cover over a defined time period $\zeta \in Z$ is now simply the number of requests it receives at each endpoint in this time period, modeled as a set of tuples $w_s^\zeta = \{\langle s_{e1}, n_1\rangle, \langle s_{e2}, n_2\rangle, \ldots, \langle s_{en}, n_n\rangle\}$, with $n_i \in \mathbb{N}^+$ representing the request count of this type in the time period. However, given the highly interconnected network structure of Microservice-based applications, each request $r \in R$ often triggers a cascade of other requests in the application. This *service call graph* is specific for a request type, and can be visualized as directed acyclic graph (DAG). An example service call graph for a request to `frontend/landing` (i.e., loading the landing page) is given in Figure 6. The semantics of the edge weights in the graph are explained below.

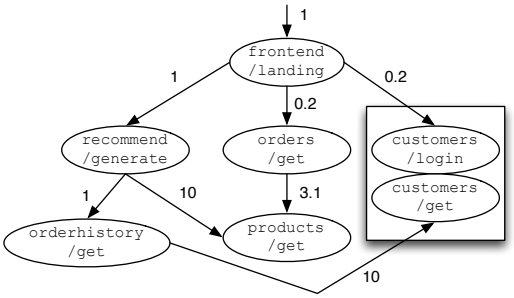

Figure 6: An example service call graph for requests to `frontend/landing`.

Formally, the service call graph associated to a request to a given endpoint $e_{scg} \in CG$ can be modeled recursively as a set of outgoing requests $\{r_{e,1}, r_{e,2}, \ldots r_{e,n}\}$. Each outgoing request $r_{e,i}$ is a tuple $\langle e, p\rangle$, with $e \in E$ and $p \in \mathbb{R}^+$. p can be understood as the stochastically expected number of outgoing subrequests per single incoming request for this endpoint. $p < 1$ represents the case where a subrequest is required only for a subset of incoming

requests (e.g., the subrequest is triggered in a branching statement in the endpoint implementation). $p = 1$ represents a one-to-one mapping, where each incoming call results in one outgoing call. $p > 1$ is the case where for every incoming request, on average more than one outgoing request is triggered (e.g., through a loop in the endpoint implementation). In the weighted DAG in Figure 6, edges are represented by the e elements, while the p elements are represented via edge weights.

2.4.2 Service Cost Model

Each service $s \in S$ further causes total operations costs in a given time period which are ultimately dependent on the number of requests that need to be handled. In CostHat, this is represented via service cost functions c, defined as in Equation 1. A cost function calculates the total deployment costs of a given service s in a defined time period ζ.

$$c(s, \zeta) : S, Z \rightarrow \mathbb{R}^+ \tag{1}$$

The total costs of the system $C(\zeta)$ for a defined time period are then defined as the sum of all service costs (Equation 2). The concrete form of a service cost function depends on whether the service is lambda- or instance-backed.

$$C(\zeta) = \sum_{s \in S} c(s, \zeta) \tag{2}$$

Lambda-Backed Microservices. For lambda-backed services, the costs of each individual endpoint is independent of all other endpoints of the service. Hence, the per-service costs are simply the sum of costs of all endpoints, as in Equation 3.

$$c(s, \zeta) = \sum_{\langle s_e, n \rangle \in w_s^\zeta} c^\lambda(s_e, n) \tag{3}$$

For each endpoint we consider cost factors $c_f \in CF_\lambda$, that linearly scale with the load parameters of the endpoint: $CF_\lambda = \{c_{API}, c_{IO}, c_{CMP}\}$. These represent the per-request API costs, compute costs, and costs of disk IO, as introduced in Section 2.2. The cost for these factors is defined as $c^\lambda_{c_f}(e) : E \rightarrow \mathbb{R}^+$. The deployment costs for an endpoint of a lambda-based Microservice are linearly dependent on the number of requests, as defined in Equation 4. c_ξ denotes other associated cost factors with operating an endpoint that are constant in nature (i.e., do not scale with load), such as fixed costs for elastic IP addresses.

$$c^\lambda(e, n) = n * \left(\sum_{c_f \in CF_\lambda} c^\lambda_{c_f}(e) \right) + c^\lambda_{c_\xi}(e) \tag{4}$$

Instance-Backed Microservices. For instance-backed services, a central notion to establish the per-service costs is the total load on the service over all endpoints. To this end, we define a per-request load factor for each endpoint $e_l \in [0..1]$. This load factor represents the percentage of a single backing instance over a time period ζ that handling a request to endpoint $e \in E$ will occupy. Using this concept, we derive the total number of backing instances required for a service during a specific time period $s_{|b|}(\zeta)$, with $s_{|b|} : Z \rightarrow \mathbb{N}^+$ in Equation 5. This function can be understood as the number of backing instances that this service's autoscaling group needs to acquire during ζ to cover the total workload over all the service's endpoints.

$$s_{|b|}(\zeta) = \left\lceil \sum_{\langle s_e, n \rangle \in w_s^\zeta} s_{e_l}\ n \right\rceil \tag{5}$$

With this information, we can now define a cost function for an instance-backed Microservice (Equation 6).

$$c(s, \zeta) = c_{inst}(s)\ s_{|b|}(\zeta) + \sum_{\langle s_e, n \rangle \in w_s^\zeta} \left(n * \Big(\sum_{c_f \in CF_\iota} c^\iota_{c_f}(s_e) \Big) + c^\iota_{c_\xi}(s_e) \right) \tag{6}$$

The cost components for API calls and disk IO are analogously to lambda-backed Microservices. We denote these as $c_f \in CF_\iota$, with $CF_\iota = \{c_{API}, c_{IO}\}$. These are again defined as functions $c^\iota_{c_f}(e) : E \rightarrow \mathbb{R}^+$. $c^\iota_{c_\xi}(e)$ again represents a constant cost term that is independent of the load on the endpoint. The main difference of instance-backed services to lambda-backed ones is that the compute costs are now defined as the number of instances that need to be running times the per-instance costs. These are defined by $c_{inst}(s) : S \rightarrow \mathbb{R}^+$. Note that this factor is service-dependent as different services may be hosted using different instance types (e.g., `m4.medium` or `c2.xlarge` in AWS EC2).

2.4.3 Calculating Total Costs

Combining the per-request service call graph models as defined in Section 2.4.1 and the service cost models defined in Section 2.4.2 now allows us to calculate the total deployment costs for a system in a time period ζ. As a precondition, we need to establish the expected *inward-facing workload*, that is how many requests to each $e \in E$ are sent to the application from outside the system, e.g., are triggered by end users by loading a Web site. These inward-facing requests are functionally identical to regular workload, and are hence described analogously as two-tuples w'^ζ_s for each service. This parameter needs to be either predicted based on previous workloads, or assumed for what-if analysis (see Section 4). We use w'^ζ to denote the collection of all inward-facing workloads in a time period.

Using this inward-facing workload, we can now calculate the resulting workload, and consecutively $C(\zeta)$. Note that the service call graph model associated to an endpoint e_{scg} readily gives us access to the information how many subrequests to which other endpoint are expected to be triggered by an incoming request. Recalling the example in Figure 6, every invocation of the endpoint `frontend/landing` triggers a single request to the endpoint `recommend/generate` and `orderhistory/get`, 0.2 requests to `orders/get` and `customers/login`, and 10 requests to `customers/get`. The endpoint `products/get` is invoked from two different services, and hence receives an expected $0.2 \cdot 3.1 + 1 \cdot 10 = 10.62$ requests per single invocation of the landing page `frontend/landing`.

This simple recursive basic principle is formalized in Listing 1 to specify how $C(\zeta)$ can be calculated assuming a given inward-facing workload. The listing uses a Python-style syntax, but applying the same terminology as used throughout this section. However, for two-tuples, we use a Python-style dictionary syntax for clarity of expression (e.g., $w'^\zeta_s[e].n > 0$). The exact calculation of $C(\zeta)$ once the workload on all services w^ζ is established is not shown in the listing, as it follows immediately from the definitions in Section 2.4.2. Note that this algorithm assumes service call graphs to be cycle-free.

```
def calc_total_costs(w'^ζ):

  '''This function calculates C(ζ)
  for an inward-facing workload w'^ζ.'''
  # Step 1: calculate total workload
  init_empty(w^ζ)
  for s ∈ S:
    for e ∈ s_e:
      if w'^ζ_s[e].n > 0:
        propagate_workload(
          e, w'^ζ_s[e].n, w^ζ
        )

  # Step 2: use Eqn 2 and following
  # to calculate total costs
  C(ζ) = 0
  for s ∈ S:
    C(ζ) += service_cost(s, ζ)

  return C(ζ)

def propagate_workload(e, n, w^ζ):

  '''This function updates w^ζ for
  the subrequests triggered by n
  additional requests to endpoint e.'''
  w^ζ_{e_s}[e].n += n
  for er in e_scg:
    propagate_workload(
      er.e, n * er.p, w^ζ
    )

def init_empty(w^ζ):

  '''Initialize the workload data
  structure empty'''
  for s ∈ S:
    for e ∈ s_e:
      w^ζ_s[e].n = 0
```

Listing 1: Calculating the per-endpoint workload $C(\zeta)$ for a defined inward-facing workload.

3. IMPLEMENTATION NOTES

Technically implementing the CostHat model in code is straightforward given the formalisms discussed in Section 2. As part of our research, we have produced two separate implementations of the model, in Java (and integrated into the Eclipse Integrated Development Environment, IDE), and in Python. The Java implementation is simplified, as it represents an earlier version of CostHat. Details to this implementation are available in a different publication [30]. The Python implementation represents the version of the CostHat model presented here. This implementation is available online[2], and forms the basis of the experimentation in Section 5. The Python implementation is a standalone tool that is not integrated into any IDE. CostHat models and workloads are provided to the tool in an XML language or via Python program code.

3.1 Instantiating CostHat for an Application

In addition to the pure implementation of the model, actually using CostHat requires developers to instantiate the model on their own, for their specific Microservice-based application. This includes defining services, endpoints, call graphs, cost models, and workloads. A comprehensive discussion of how these various model parameters can be established for applications written using different technology and operated in different environments is out of scope for this paper. However, we briefly discuss strategies, existing work, and tools that can serve as a starting point in Table 1.

[2]https://github.com/xLeitix/costhat

4. USING COSTHAT MODELS

After instantiating the CostHat model for an application, it can be used as basis for a variety of analyses and developer tool support. In the following, we present three examples without claim of completeness.

4.1 Raising Developer's Cost Awareness

Even independently of any deeper analysis, having access to a CostHat model alone may already raise the cost awareness of developers that work on services in a Microservice-based application. For instance, we have have observed in a previous study that developers deem cloud deployment costs to be an important, but often intangible, metric [7]. Our work can help make costs more explicit. For instance, using CostHat, developers are enabled to explicitly see what costs each service and endpoint is contributing to the total deployment costs, as well as why (i.e., due to which interactions). CostHat can also be used as basis for establishing per-customer deployment costs, supporting business decisions on service pricing.

4.2 What-If Analysis

A core functionality enabled by CostHat are *what-if analyses*. Essentially, once all parameters of the model are established (see also Table 1), developers can vary individual parameters to identify the cost impact of changes. This enables developer support tooling that can be used to, for instance, answer the question how the costs would increase under increased inward-facing workload (varying w'^{ζ}_s), increased cloud instance prices (varying $c_{inst}(s)$), or migration to a different cloud provider or instance type for an instance-backed Microservice (varying all cost parameters). Further, using what-if analysis, developers can also decide whether improving the implementation of a service is worth the effort in terms of costs, by analyzing the cost impact of a slightly lower e_l for an endpoint implementation. All of these analyses allow data-driven decision making [2] in software development.

Figure 7: Example of an Eclipse IDE integration of what-if analysis using CostHat.

We have implemented an example of some of these analyses as part of our Eclipse-based prototype for an earlier version of CostHat. A core property of this example implementation is that it integrates tightly with standard development tools, and allows

Parameter	Description
Services and their endpoints (S, E)	For small service-based systems, the information which services are available and which endpoints they provide may already be well-known to developers [26]. However, even if this is not the case, service repositories such as VRESCo [24] may be used to automatically produce a service model.
Service call graphs for all endpoints (SCG)	The service call graph model described in Section 2.4.1 can essentially be established in two ways: (1) post-hoc, by mining call traces and logs of previous executions of the application; while technically not trivial, tools that support this kind of service dependency mining exist (e.g., Dynatrace); (2) prior to deployment, by statically analysing the source code implementing service endpoints for invocations of other services in the system. In practice, a combination of both methods seems most promising. Mining execution logs allows to relatively easily identify invocations across heterogeneous service implementations, while static analysis enables the detection of rarely invoked dependencies. Further, detecting node weights in service call graphs is not necessarily possible using static code analysis alone.
Per-request cost parameters per service and endpoint (c_{API}, c_{IO}, and c_{CMP} for lambda-backed services)	Assuming some historical executions of the application, all of these cost parameters are readily available from common cloud monitoring solutions (e.g., Amazon CloudWatch), or can be derived from cloud monitoring data and cloud provider cost specifications with little difficulty.
Constant costs per service $(c^{\iota}_{c_\lambda}, c^{\iota}_{c_\xi})$	The request-independent costs per service are typically known to the service provider. Hence, this parameter requires no specific technology support.
Load factors per endpoint of instance-backed service (e_l)	Similar to service call graphs, existing application performance monitoring (APM) solutions such as Dynatrace can be used to establish the average number of concurrent requests of a specific endpoint a single backend instance can handle.
Instance costs per instance-backed service $(c_{inst}(s))$	This parameter is readily available from cloud provider cost specifications.
Inward-facing workload (w'^{ζ}_s)	While it is evidently not possible to foresee the inward-facing workload for a future time interval exactly, DevOps engineers and operators of existing services tend to have reasonably accurate (explicit or implicit) prediction models for expected workloads. Further, this parameter is often used as the basis for what-if analysis anyway (see also Section 4.2).

Table 1: Summary of CostHat parameters and basic implementation strategies.

for what-if analysis directly in the source code. This avoids mentally taxing context switches, and is conceptually similar to the idea of feedback-driven development (which we have previously presented [8]). An example screenshot that illustrates the idea is given in Figure 7.

4.3 Predicting Costs of Code Changes

A CostHat use case that is somewhat more elaborated than what-if analysis is the pre-deployment prediction of the impact of given code changes on the deployment costs. The fundamental idea here is similar to what-if analysis, but rather than varying workload or cost parameters, we now analyze the impact of changes in the service call graph model. This includes adding a new edge to the service call graph (predicting the impact of adding a new service invocation), removing edges (removing service invocations), or moving a service implementation into a loop.

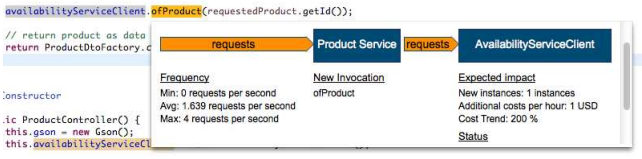

Figure 8: Example of a prototypical Eclipse IDE integration that implements cost prediction for code changes based on CostHat.

Our Eclipse-based prototype contains a simplified version of this idea (Figure 8). The prototype is able to discover when a code change would introduce a new service invocation via static code analysis, recalculates the CostHat model in the background, and produces a code-level warning (including detailed analysis screen, which leads back to a what-if analysis) if the predicted added cost are above a defined threshold.

5. EVALUATION

We showcase and evaluate the CostHat model in two dimensions. Firstly, we illustrate the basic idea behind the model with a simple Python-based implementation and using the example introduced in Figure 1. Secondly, we numerically show that it is possible to quickly (re-)evaluate the costs of even large Microservice-based applications using CostHat, enabling real-time service developer tools, such as the ones envisioned in Section 4.

5.1 Case Study

We assume that the case study application is deployed in AWS, using a combination of EC2 and Lambda services. The endpoint `banking_interface` is an external service that is charged on a per-use basis. We model the CostHat parameters of this case based on our previous experience benchmarking Web applications in AWS [3, 17]. For pricing (e.g., EC2 hourly instance prices, or Lambda per-request costs), we assume real-life AWS pricing for North America at the time of writing, and for on-demand instances. In total, this case study consists of 7 services and 11 endpoints. The CostHat model for the case is publicly available on GitHub along

	frontend	recommend	orders	products	customers	order_hi	banking_if	**Total**
Baseline	3.35$	11.51$	3.24$	1.04$	18.32$	21.85$	13.54$	**72.85$**
Scenario 1	6.37$	11.51$	6.36$	1.09$	18.32$	36.85$	27.09$	**107.59$**
Scenario 2	3.35$	15.34$	3.24$	1.30$	18.32 $	21.85$	13.54$	**76.95$**
Scenario 3	3.35$	11.51$	3.24$	1.04$	18.32$	21.48$	13.54$	**72.48$**
Scenario 4	3.35$	11.51$	3.24$	1.04$	18.84$	21.85$	13.54$	**73.38$**

Table 2: CostHat deployment cost calculations in US dollars per hour for the case study. The baseline represents the deployment costs of the application as modeled, while the four scenarios represent various illustrative changes and analyses.

with the Python implementation of the model.

Table 2 provides the total hourly deployment costs as calculated by CostHat. Firstly, we evaluate the costs for the application as modelled. Afterwards, we conduct what-if analysis on the model as discussed in Section 4.2, and evaluate the following four scenarios. It should be noted that these scenarios represent examples rather than a comprehensive list of analyses that are enabled by CostHat.

- **Scenario 1 – Improved conversion rate.** This scenario assumes that the Web shop is able to improve its conversion rate, i.e., that twice as many users that use the service endpoint `frontend/landing` also end up using the endpoint `frontend/order`. Evidently, the increased conversion rate has ripple effects through the entire application, leading to substantially higher deployment costs. While improving the conversion rate may still make business sense, the responsible DevOps engineers will need to keep these effects in mind.

- **Scenario 2 – External pressure on the recommender engine.** This scenario evaluates what would happen if the service `recommend` would start to be used in a different application as well, leading to substantial additional external pressure. This additional workload unsurprisingly leads to higher costs for hosting the `recommend` service, but dependencies of this service get slightly more expensive as well.

- **Scenario 3 – Instance type migration for the orderhistory service.** In this scenario, the DevOps engineers are evaluating whether to migrate the `order_history` service from using expensive `i2.xlarge` instances to cheaper `m4.2xlarge` instances. The analysis shows that all things considered CostHat predicts the costs to remain close to constant after the migration. Hence, the DevOps engineers may decide that conducting this migration is not worth their time in the current setup.

- **Scenario 4 – Impact of code change.** This scenario illustrates the idea described in Section 4.3, and evaluates the cost impact of a planned code change that introduces a new dependency of `recommend/generate` to `customers/get`. The analysis shows that this additional dependency will lead to a modest 0.90$ per hour additional costs. In isolation, this is likely unproblematic, however, the team will need to stay aware of the added costs of code changes, as such small expenses may add up over time if many dependencies are added.

5.2 Experimentation

This experiment aims to address the question how computationally expensive (re-)evaluating the CostHat model is. To this end, we evaluate the time necessary for cost computation for automatically generated Microservice networks, for which we gradually increase network size (in number of services and endpoints) and connectedness (in number of endpoint-to-endpoint connections, i.e., service calls).

5.2.1 Experiment Setup

As basis for this evaluation, we use a script, which is also available on the GitHub page that hosts the Python implementation, to artificially generate CostHat models. We evaluate two scenarios. In the *network size* scenario, we gradually increase the number of nodes in the network, and observe how the time necessary to calculate the total deployment costs depends on this parameter. In this scenario, we assume that each endpoint has between 0 and 10 outgoing service calls. In the *connectedness* scenario, we keep the number of nodes in the network fixed to 5000, but vary the total number of edges. We repeat each scenario for both, lambda- and instance-backed services. All other parameters (e.g., c_{API} or c_{IO}) remain fixed through all experiments, as they do not influence the time necessary to calculate costs. The basic parameters of our experiments are summarized in Table 3.

	Network Size Scenario	**Connectedness Scenario**
Nodes (Services and Endpoints)	[10..8000]	5000
Edges (Service Calls)	[0..10] per node	[100..8000] total

Table 3: Basic parameters of the experiment setup.

All experiments have been conducted on a standard Macbook Pro with a 2.8 GHz Intel Core i7 processor and 16 GB RAM. The test machine was running OS X El Capitan.

5.2.2 Results

Figure 9 depicts the results for the network size scenario. The y-axis shows the time in milliseconds necessary to evaluate the total deployment costs of the model 100 times. First and foremost, we observe that the total time necessary to evaluate the costs of an application using CostHat is very low. Even for large service networks, calculating one single cost model never takes more than 8 milliseconds on the test machine (broken down from 100 calculations). For networks at or below 1000 nodes, the re-calculation time stays below 1 millisecond. In addition, we observe that for both, lambda- and instance-backed services, the time necessary to calculate costs linearly depends on the number of nodes in the network. The costs of lambda-backed services can be evaluated faster than the costs of instance-backed services.

Figure 10 shows the results for the connectedness scenario. For instance-backed services, we again observe a linear dependency between the total number of service calls in the application, and the time necessary to calculate the total deployment costs. For lambda-backed services, no clear dependency could be established in these

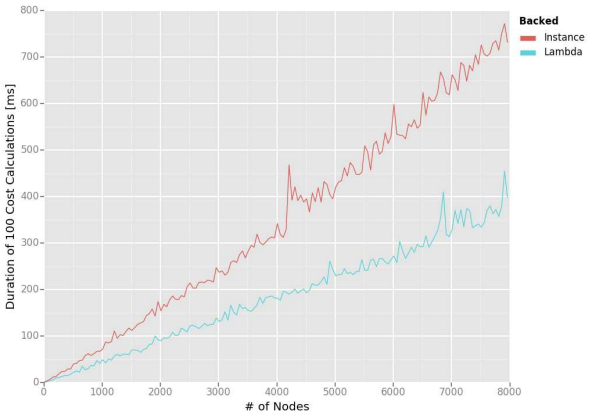

Figure 9: Time necessary to calculate the total costs of a CostHat model 100 times depending on the number of services and endpoints in the model.

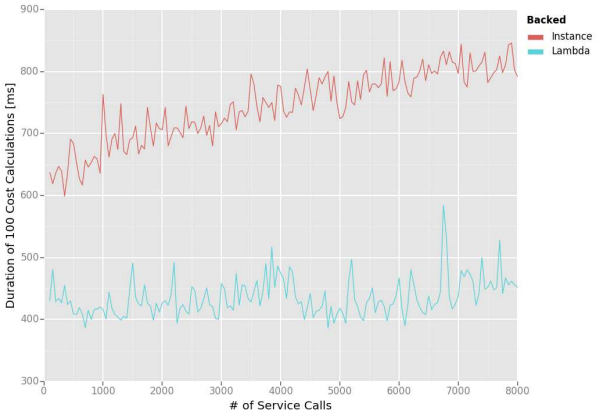

Figure 10: Time necessary to calculate the total costs of a CostHat model 100 times depending on the total number of service calls in a network of 5000 services and endpoints.

experiments. However, it is evident for both types of services that the cost calculation duration depends more on the number of services and endpoints in the system than on the sizes of the service call graphs.

6. RELATED WORK

Costs in Microservice-based applications arise as soon as incoming requests trigger computation within the service, or as soon as cloud resources are allocated in case of instance-backed Microservices. As a consequence, our work bears some similarities to research concerned with cost-efficient cloud resource management (e.g., [15]). As the demand for cloud resources is often highly variable, a plethora of research aims at predicting future demand, for instance future workload (capacity planning). However, these research endeavours ground in a fundamentally different objective, namely, to make allocation, provisioning and scheduling decisions, either on cloud provider side to minimize monetary cost while providing a viable service [9, 12, 31], or on cloud user side to help solving various optimization problems. These include cost minimization or performance maximization problems for a given workload [23, 31], cost trade-off decisions between scaling up/out or accepting to delay incoming requests [18, 29] (which often includes the challenge of monitoring SLAs [10]), or modelling budget constraints and time constraint decisions [19–21].

In contrast to this research, CostHat has its focus on Microservices which are permanently running in the cloud. We further differ fundamentally from the previously mentioned work in that we consider the provisioning of cloud resources as a blackbox, which occurs automatically based on an autoscaling unit keeping the quality in terms of response time and throughput constant. Still, there seems to be a general shift from reactive towards proactive cloud resource management, whereof our proposed use case of increasing developerâĂŹs awareness about their actual and potential future cloud resource costs may be seen as an extreme form of proactive cloud resource management and a next step towards building cloud-native applications [1].

Furthermore, our model has similarities to the prototype proposed in [28], which enables what-if analysis in complex distributed data center applications. They also use a graph-based approach to model service calls and response times (in addition to harnessing queuing theory). By offering a custom query language they facilitate a what-if analysis of hypothetical changes in the applicationâĂŹs workload or environment and estimate its impact on performance. In contrast, we link the service call graph with service cost models. This novel combination serves as a basis to predict the cost impact of a certain potential (what-if) or actual change of the invocation pattern of a certain service endpoint.

However, the problem of collecting tracing data of invocations between individual service (endpoints) is more general as it is required for a variety of endeavours, (e.g., performance monitoring, QoS monitoring, and others). Therefore, a vast body of knowledge is available about how to generate service call graphs automatically. Dynamic approaches that collect data after deployment and during production are widely discussed in research (e.g., [24]), by companies such as Google [27], and have led to a variety of tools (e.g., Dynatrace[3]). In this paper, we primarily use a simplified approach based on probabilistic expected values to model communication patterns in the service call graph. A broad range of related work discussed models of incoming requests based on statistical time series analysis using Fast Fourier Transform (FFT) and Markov chains [11], autoregressive models [6, 16], linear regression, and neural networks [14].

Furthermore, Microservice-based applications as a collection of independent services can be seen as a pragmatic implementation of service-oriented systems [13]. However, to the best of our knowledge, this is the first approach that brings together service call trace data with cloud cost models. Being able to link propagation patterns of calls between service endpoints with costs, provides a unique basis for a variety of analyses, whereof a few have been discussed in Section 4.

One of the use cases of the CostHat approach is within cloud developer tooling directly in the IDE, which allows for cost analysis of code changes without requiring a context switch. This establishes a connection to research to Feedback-Driven Development [4, 8]. However, it is crucial to note that the use of the proposed CostHat model is not limited to pre-deployment tooling. To date, most cost monitoring applications offered by cloud providers (e.g., Amazon Cloud Watch[4]) provide cost information on an instance- or service-level. In contrast, CostHat dives deeper into details as the

[3]http://www.dynatrace.com

[4]https://aws.amazon.com/cloudwatch/

cost data is linked to the service call graph.

7. CONCLUSIONS

In this paper, we propose a graph-based model of the deployment costs of Microservice-based applications. The CostHat model can be used for applications that use canonical instance-backed Microservices, as well as ones that are implemented on top of AWS Lambda or similar services. CostHat models costs caused by IO, compute, API calls, and other constant cost factors, and can model the total deployment costs depending on the call patterns between services in the Microservice-based application. In addition to the basic model, we have also presented initial ideas on how to extract a CostHat model from a production application. Further, we have illustrated use cases of CostHat through two example implementations, one integrated into the Eclipse IDE and another standalone in Python. We also provide this Python implementation of the model as open source software.

We have evaluated our work based on a small case study and via scalability experiments using the Python implementation. The case study has illustrated that CostHat can be used in a variety of developer and DevOps support scenarios. The conducted scalability experiments have shown that evaluating even large service networks is computationally cheap, typically in the sub-microsecond range. Hence, value-added tools such as the ones proposed in Section 4, which require constant re-evaluation of CostHat models to give developers real-time feedback directly in their IDE, are possible.

7.1 Future Research Directions

The presented work requires future work in two directions. On the one hand, it is necessary to better support or automate the process of instantiating a CostHat model from a concrete application, as discussed in Section 4. Tools are needed that are able to (semi-)automatically analyze logs to identify services and endpoints, or to construct service call graphs. Further, establishing concrete per-request IO API costs, or load factors for instance-backed services is not necessarily easy for Devops engineers, and requires further technical and conceptual support. On the other hand, we plan to conduct more work on concrete developer-facing tools that build on top of CostHat, such as the examples already given in Section 4. In order to do so, studies are needed that investigate which deployment cost related questions developers and DevOps engineers typically need to answer, so as to provide the right kind of tooling.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss National Science Foundation (SNF) under project MINCA (Models to Increase the Cost Awareness of Cloud Developers).

8. REFERENCES

[1] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, May 2016.

[2] A. Begel and T. Zimmermann. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 12–23, New York, NY, USA, 2014. ACM.

[3] A. H. Borhani, P. Leitner, B. S. Lee, X. Li, and T. Hung. WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines. In *Proceedings of the 2014 IEEE 18th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 101–109, Sept 2014.

[4] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, and C. Woods. CloudWave: where Adaptive Cloud Management Meets DevOps. In *Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014)*, 2014.

[5] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computing Systems*, 25:599–616, 2009.

[6] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers using Online Measurements. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):300, 2003.

[7] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA, 2015. ACM.

[8] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth. Runtime Metric Meets Developer - Building Better Cloud Applications Using Feedback. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*, New York, NY, USA, 2015. ACM.

[9] J. Doyle, V. Giotsas, M. A. Anam, and Y. Andreopoulos. Cloud Instance Management and Resource Prediction for Computation-as-a-Service Platforms. In *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E*, pages 89–98, Apr. 2016.

[10] A. García García, I. Blanquer Espert, and V. Hernández García. SLA-Driven Dynamic Cloud Resource Management. *Future Generation Computing Systems*, 31:1–11, 2014.

[11] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for Cloud Systems. *Proceedings of the 2010 International Conference on Network and Service Management (CNSM 2010)*, pages 9–16, 2010.

[12] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Computing Communications Review*, 39(1):68–73, Dec. 2008.

[13] M. N. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, Jan. 2005.

[14] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computing Systems*, 28(1):155–162, 2012.

[15] B. Jennings and R. Stadler. Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, 23(3):567–619, 2 Mar. 2014.

[16] G. Jung, M. a. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. *Proceedings of the International Conference on Distributed Computing Systems*, pages 62–73, 2010.

[17] P. Leitner and J. Cito. Patterns in the Chaos – a Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology*, 2016.

[18] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 213–220. ieeexplore.ieee.org, June 2012.

[19] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and Deadline-Constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, page 22, Los Alamitos, CA, USA, 10 Nov. 2012. IEEE Computer Society Press.

[20] M. Mao and M. Humphrey. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 67–78. ieeexplore.ieee.org, May 2013.

[21] M. Mao, J. Li, and M. Humphrey. Cloud Auto-Scaling with Deadline and Budget Constraints. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48. ieeexplore.ieee.org, Oct. 2010.

[22] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011.

[23] R. Mian, P. Martin, and J. Vazquez-Poletti. Provisioning Data Analytic Workloads in a Cloud. *Future Generation Computing Systems*, 29(6):1452–1458, 2013.

[24] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Binding, and Mediation in VRESCo. *IEEE Transactions on Services Computing*, 3(3):193–205, July 2010.

[25] S. Newman. *Building Microservices*. O'Reilly, 2015.

[26] G. Schermann, J. Cito, and P. Leitner. All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. In *Proceedings of the 11th International Workshop on Engineering Service Oriented Applications (WESOA'15)*, 2015.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[28] R. Singh, P. Shenoy, M. Natu, V. Sadaphal, and H. Vin. Predico: a System for What-If Analysis in Complex Data Center Ppplications. In *Proceedings of the 12th International Middleware Conference*, pages 120–139, 2011.

[29] M. Smit and E. Stroulia. Configuration Decision Making Using Simulation-Generated Data. In E. Michael Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig, and M. Fantinato, editors, *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, Lecture Notes in Computer Science, pages 15–26. Springer Berlin Heidelberg, 7 Dec. 2010.

[30] E. Stöckli. Feedback Driven Development – Predicting the Costs of Code Changes in Microservice Architectures based on Runtime Feedback. Master's thesis, University of Zurich, 2015.

[31] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible Use of Cloud Resources Through Profit Maximization and Price Discrimination. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 75–86, Apr. 2011.